Università di Bologna, Sede di Cesena
Ingegneria e Scienze Informatiche LM

Sistemi Distribuiti

REST Best Practices

Alessandro Maretti alessandro.maretti@studio.unibo.it

Sofia Rosetti sofia.rosetti@studio.unibo.it

20/03/2018

`https://github.com/SofiaRosetti/SD1617_RestFul`

Indice

Capitolo 1

Introduzione

In questa relazione si vuole presentare una raccolta di best practices riguardanti la struttura e l'utilizzo di una piattaforma RESTful basandosi su un progetto fornito dal docente per gli esempi e la messa in pratica di tali procedure utilizzando la libreria Jersey. E' stata inoltre inserita una panoramica generale sui servizi RESTful e sulla continuous integration, per la quale è stato creato un repository gitHub.

Capitolo 2

I web services

> I Web Services sono sistemi software progettati per supportare l'interoperabilità tra diversi elaboratori su una medesima rete, ovvero in un contesto distribuito.

Dalla definizione tratta dal W3C deduciamo che questi permettono quindi di far interagire in maniera trasparente applicazioni sviluppate con linguaggi di programmazione diversi, che girano su sistemi operativi eterogenei realizzando porzioni di funzionalità in maniera indipendente e su piattaforme potenzialmente incompatibili, facendole interagire tramite tecnologie Web e creando un'architettura facilmente componibile e scalabile.

Attualmente esistono due tecnologie per lo sviluppo di Web Services:

- SOAP (Simple Object Access Protocol): un protocollo per lo scambio di messaggi e l'invocazione di servizi remoti;

- REST (REpresentational State Transfer): può essere descritto come un'architettura indipendente dalla tecnologia e dalla piattaforma utilizzata, dove i componenti comunicano tramite interfacce basate su standard web come HTTP in cui le funzionalità ed i dati sono considerati come risorse raggiungibili tramite URI ed azionabili tramite operazioni ben precise (GET, POST, PUT, DELETE, HEAD e OPTIONS). L'intera architettura è strutturata per massimizzare l'efficienza e la scalabilità rendendolo quindi ottimo per software distribuiti.

Tuttavia la definizione esatta di REST resta (ad oggi) inesistente in quanto, in base alla fonte dalla quale si reperiscono le informazioni, viene descritta come una tecnologia, un'architettura o un framework. La terminologia forse più corretta per definire REST è definirlo come "un insieme di regole generiche" riguardanti la struttura logica di un'applicazione (anche non necessariamente web).

In questo progetto sono stati implementati servizi web RESTful, basati quindi sulle linee guida di REST e su SOA (resource-oriented architecture), un'architettura software e paradigma di programmazione per il design e lo sviluppo di software in forma di risorse, ovvero componenti del software che possono essere riusati per differenti scopi.

2.1 Struttura di un'applicazione RESTful

Il concetto centrale di un'applicazione RESTful è quello di risorsa, ovvero qualunque entità che possa essere indirizzabile tramite web in maniera univoca ed accessibile/trasferibile tra client e server tramite una rappresentazione dello stato interno della risorsa stessa. La rappresentazione può essere basata su vincoli diversi, per cui JSON e XML possono rappresentare lo stesso modello, ma qualunque sia la rappresentazione questa deve essere in grado di supportare i link.

Supportare differenti rappresentazioni può portare a problemi di interpretazione: nel caso in cui un nuovo client cambi i requisiti per i modelli dei dati in maniera non retro-compatibile il servizio potrebbe non riuscire a rispondere. Per questo motivo è stata introdotta la Content Negotiation, secondo la quale i formati di dati specifici da accettare o restituire da parte del service sono negoziati a runtime come parte dell'invocazione (Media Types). Tale procedimento non richiede l'invio di ulteriori messaggi ma il client invia semplicemente i formati comprensibili (MIME types) ed il server sceglie il formato più adatto per la risposta.

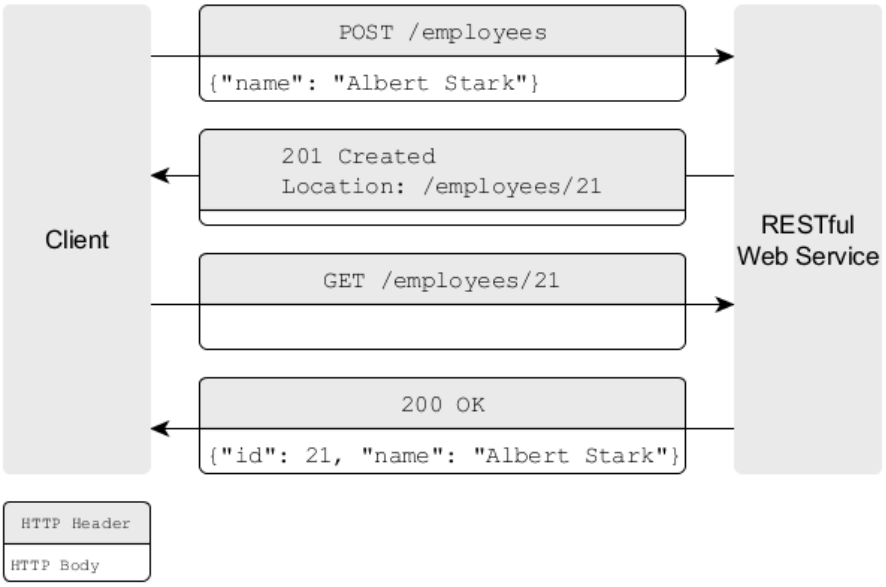
Infine i fattori di qualità permettono al client di indicare il grado di preferenza per ogni rappresentazione utilizzando valori da 0 a 0.9.
La gestione delle risorse avviene tramite i principali comandi del protocollo HTTP (citati nel capitolo precedente) che identificano differenti azioni da effettuare sulle risorse, nella tabella sotto indicata vengono raccolti

tutti i metodi, con una breve descrizione e l'appartenenza a metodi idempotenti, ovvero in cui la richiesta può essere ripetuta senza generare effetti sullo stato della risorsa (se ad esempio il server si blocca la richiesta può essere ripetuta fino al corretto esito senza problemi) oppure safe, in cui la richiesta non genera effetti collaterali sul server (le richieste unsafe invece generano effetti sul server):

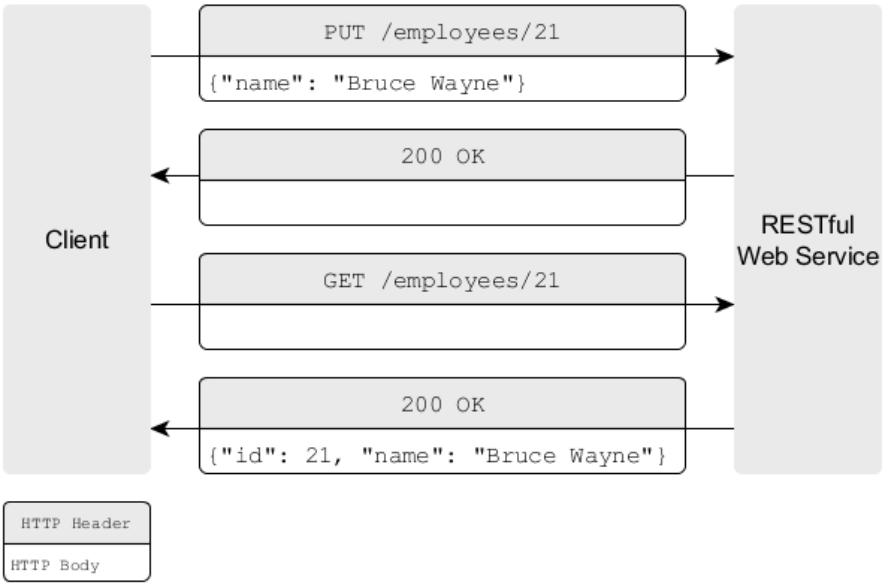
Metodi REST

Metodo	Semantica	Idempot.	Safe
POST	Crea una risorsa subordinata ad una esistente	NO	NO
GET	Restituisce una rappresentazione della risorsa	SI	SI
PUT	Inizializza o aggiorna lo stato di una risorsa	SI	NO
DELETE	Elimina una risorsa	SI	NO
HEAD	Legge i metadati di una risorsa (GET senza response body)	SI	SI
OPTIONS	Elenco di operazioni possibili su una risorsa	SI	SI
PATCH	Modifica una parte di una risorsa	SI	NO

Esempio di messaggio POST per la creazione di una risorsa.



Esempio di messaggio PUT per la modifica di una risorsa.



Essendo la risorsa un concetto centrale all'interno di un'architettura di tipo RESTful, si può affermare che sia un'architettura di tipo ROA (Resource-Oriented Architecture). Unendo i principi alla base di REST e alla base di ROA è possibile affermare che questo tipo di architettura avrà alla base cinque principi:

- Identificazione delle risorse: ogni risorsa dovrà essere identificabile tramite URI (Uniform Resource Identifier) in maniera univoca e universale.
- Interfacce uniformi: lo stesso piccolo insieme di operazioni si applica a tutto, si utilizzano pochi verbi (citati in precedenza) per un insieme

esteso di sostantivi.

- Self-Descriptive Messages: ogni messaggio contiene le informazioni necessarie per la propria gestione.

- Hypermedia come motore dell'applicazione: le rappresentazioni delle risorse che vengono trasferite contengono link, per cui si può affermare che le applicazioni RESTful "navigano" piuttosto che "chiamare" poiché procedono sulla base dei link inseriti nella rappresentazione delle risorse.

- Iterazioni stateless: ogni richiesta dal client al server deve contenere le informazioni necessarie per capire completamente la richiesta, indipendentemente da qualunque richiesta precedente ed effettuando la richiesta in completo isolamento.

Per la realizzazione di servizi RESTful normalmente si segue una metodologia di progettazione per le applicazione simile a quella che segue:

1. si devono identificare tutte le risorse che devono essere esposte come servizio;

2. si modellano tutte le relazioni tra le risorse con collegamenti ipertestuali che possono essere seguiti per ottenere maggiori dettagli o transizioni di stato;

3. si definisco URI corretti per indirizzare le risorse;

4. si cerca di determinare cosa comporta l'utilizzo dei metodi HTTP su ogni risorsa;

5. si progetta e si documenta la rappresentazione delle risorse.

Capitolo 3

Le best practices

Nello sviluppo di applicazioni RESTful è consigliabile seguire alcune best practices, di seguito verranno quindi elencate, descritte e, dove possibile, verranno svolti esempi basati sul progetto fornito dal docente. In tale progetto vengono utilizzate per l'implementazione dei servizi RESTful le librerie Jersey[1]. Jersey è l'implementazione di riferimento della specifica JAX-RS (JSR 311) per la realizzazione di Web Service RESTful su piattaforma Java, inoltre fornisce le proprie API che estendono il toolkit JAX-RS con funzionalità aggiuntive che permettono di semplificare ancora di più i servizi RESTful e lo sviluppo del client.

1. Per alterare lo stato si devono utilizzare solamente i metodi POST, PUT (o eventualmente PATCH), DELETE e non il metodo GET.

[1] https://jersey.github.io/

Ad esempio, per modificare lo stato del CSV riguardante il modello gfghjgj può essere efficace sia la chiamata:

POST /hadrian/model/gfghjgj/csv

oppure, se la modifica è di una parte della risorsa:

PATCH /hadrian/model/gfghjgj/csv

2. E' preferibile utilizzare nomi al posto di verbi, in modo da ottenere una forma più intuitiva e comune. I nomi dovrebbero essere di facile comprensione da parte degli utenti che utilizzano l'API; inoltre, in alcuni framework moderni, viene utilizzata in automatico tale dicitura. Ad esempio è meglio utilizzare una sintassi come la seguente:

   GET /employees

   GET /employees/123

   POST /employees

   Piuttosto che una come questa:

   GET /getAllEmployees

   GET /getEmployeeWithId

   POST /createEmployee

3. Il formato dell'URL dovrebbe prevedere solamente l'utilizzo del plurale, in modo da facilitare l'utilizzo da parte dell'utente e l'implementazione da parte del fornitore. Ovvero, riprendendo spunto dall'esempio precedente, per far riferimento alla risorsa che riguarda un singolo impiegato lo URI più corretto sarà:

GET /employees/123

e non

GET /employee/123

Nel codice preso in esame occorrerebbe quindi apportare delle modifiche alle seguenti parti:

- @javax.ws.rs.Path("model/modelID")
  –> @javax.ws.rs.Path("models/modelID")
- @javax.ws.rs.Path("model/pfa/modelID")
  –> @javax.ws.rs.Path("models/pfa/modelID")
- @javax.ws.rs.Path("model/pfa/redis/modelID")
  –> @javax.ws.rs.Path("models/pfa/redis/modelID")
- @javax.ws.rs.Path("model/modelID/csv")
  –> @javax.ws.rs.Path("models/modelID/csv")

Per quanto riguarda le estensioni del tipo di file (vedi pfa e csv) potrebbe essere conveniente, per rispettare la sintassi, modificarli in pfaFiles e csvFiles, in modo da mantenere il plurale.

4. Dovrebbero essere presenti due URL per ogni risorsa, uno per identificare la collezione di elementi ed uno per identificare uno specifico elemento. Come visto anche poco sopra dovremmo sempre avere:

   @javax.ws.rs.Path("models")

   @javax.ws.rs.Path("model/modelID")

5. Oggi le web API sono accessibili tramite connessioni Internet che spesso non sono sicure. Molte comunicazioni non sono criptate e ciò consente attacchi come eavesdropping o hijacking di credenziali. Per questo motivo è auspicabile utilizzare sempre SSL in quanto garantisce comunicazioni crittografate e semplifica l'autenticazione (si possono utilizzare dei token di accesso invece di effettuare una nuova autenticazione ad ogni richiesta).

6. Tutte le API dovrebbero essere fornite di un'appropriata documentazione facilmente e pubblicamente accessibile. Questo consentirà agli sviluppatori di analizzare le caratteristiche dell'API prima di un'eventuale integrazione.
   Nel caso del progetto preso in esame abbiamo deciso di inserire, nell'header delle risposte del server, un campo "info" ovvero da così:

```
 @GET
@javax.ws.rs.Path("models")
@Produces({ MediaType.APPLICATION_JSON,
    MediaType.APPLICATION_XML })
public List<BlackBoxModel> listModels() {
   return this.hadrian.listModels();
}
```

a così:

```
@GET
@javax.ws.rs.Path("models")
@Produces({ MediaType.APPLICATION_JSON,
    MediaType.APPLICATION_XML })
public Response listModels() {
   Response.ResponseBuilder builder =
       Response.ok(this.hadrian.listModels());
   builder.header("Info", "list of models on sever");
   return builder.build();
}
```

7. Tutte le API dovrebbero avere uno sviluppo basato su diverse versioni. Sebbene sia possibile includere la versione dell'API anche nell'URL, è più probabile che questa sia inserita in un header. Tuttavia, è consigliabile includere la versione anche nell'URL per fare in modo che le risorse possano essere esplorate agilmente tra le varie versioni utilizzando il browser. Nel progetto la soluzione più conveniente sarebbe suddividere le versioni in vari packages in modo da semplificarne la gestione.

8. L'URL deve essere il più compatto e semplice possibile, per man-

tenere chiarezza all'interno degli URL, le operazioni di filter e sort possono esservi incluse alla fine come parametri di una query. Ad esempio, per effettuare un'operazione di ordinamento, possiamo utilizzare una sintassi come la seguente:

GET /tickets?sort=-priority

oppure, per filtrare i risultati:

GET /tickets?state=open

9. E' consigliabile seguire il principio di "HATEOAS" (Hypermedia as the Engine of Application State), che prevede di fornire alcuni link in grado di migliorare la navigazione attraverso l'applicazione, in modo che il compito di costruire gli URL non sia demandato agli utenti. L'API viene così resa più auto-descrittiva e si assicura che il client ottenga sempre un URL valido. All'interno del progetto abbiamo fatto in modo che a fronte della richiesta di un particolare modello venga restituito, oltre al modello stesso, anche il link al file CSV relativo.

```
@GET
@javax.ws.rs.Path("models/{modelID}")
@Produces({ MediaType.APPLICATION_JSON,
    MediaType.APPLICATION_XML })
```

```
public Response modelDescriptor(@PathParam("modelID")
    final UUID modelID) {
   Map map = new LinkedHashMap<String, String>();
   map.put("descriptor",
       this.hadrian.modelDescriptor(modelID));
   map.put ("linkCsv","models/"+ modelID +"/csvFiles");
   Gson gson = new Gson();
   Response.ResponseBuilder builder =
       Response.ok(gson.toJson(map));
   builder.header("Info", "list of models on sever");
   return builder.build();
}
```

10. Normalmente l'utente non necessita di visualizzare una rappresentazione completa di ogni risorsa. Si può quindi fare in modo che il traffico di rete venga minimizzato, con un conseguente incremento delle prestazioni dell'API. Una tecnica spesso utilizzata prevede di utilizzare un parametro all'interno della query che prenda la lista dei campi da includere separati da virgola.

    Ad esempio, nella richiesta sottostante si vuole ottenere l'elenco dei biglietti, ma con solo alcuni parametri, quali id, subject, customer_name e updated_at.

```
GET /tickets?fields=id,subject,customer_name,updated_at
```

11. E' consigliabile utilizzare un meccanismo di paginazione, in modo da non ritornare le risorse presenti all'interno del database in un'unica volta. I parametri maggiormente utilizzati sono offset e limit, i cui valori di default sono solitamente 0 e 10.

    All'interno del progetto in esame è stata applicata questa regola nella richiesta dell'elenco dei modelli. Essi non vengono restituiti tutti contemporaneamente, ma solo quelli compresi fra i parametri offset e limit:

    GET http://localhost:9998/hadrian/models?offset=0&limit=10

```
 @GET
@javax.ws.rs.Path("models")
@Produces({ MediaType.APPLICATION_JSON,
    MediaType.APPLICATION_XML })
public Response listModels(@QueryParam("offset") int
    offset, @QueryParam("limit") int limit) {
   List<BlackBoxModel> list = this.hadrian.listModels();
   if (limit > list.size()) {
      limit = list.size();
   }
```

```
        Response.ResponseBuilder builder =
            Response.ok(list.subList(offset, limit));
        builder.header("Info", "list of models on server");
        return builder.build();
    }
```

12. Le operazioni di creazione e aggiornamento (PUT, POST, PATCH) dovrebbero ritornare una rappresentazione della risorsa che è appena stata creata o modificata.

13. Per la creazione di nuove risorse è necessario utilizzare PUT se si vuole che il client decida l'URI della risorsa, mentre è necessario utilizzare POST se la decisione dell'URI spetta la server.

14. Dato che alcuni proxy supportano solamente i metodi GET e POST, è necessario effettuare un override dei metodi HTTP in modo da rendere utilizzabile una API RESTful anche in presenza di queste limitazioni.

15. Dove poJssibile, è preferibile l'utilizzo di JSON rispetto a XML, in quanto fare parsing o letture su quest'ultimo è più complesso.

16. Se si utilizza JSON come formato di rappresentazione è preferibile adottare la convenzione "camelCase" per i nomi dei campi.

17. La codifica degli URL non prevede il concetto di "tipo di dati" e di "strutture gerarchiche", per cui l'applicazione è costretta a considerare interi e booleani come stringhe e non è in grado di fare confronti con la struttura gerarchica nativa di JSON. Per questo motivo le API che utilizzano una codifica JSON dovrebbero richiedere che il campo "Content-Type" dell'header sia impostato come application/json. In caso contrario si ritornerà un codice di errore 415 (Unsupported Media Type).

18. In alcuni casi l'applicazione utente necessita di caricare dati relativi alle risorse che sono state richieste. Attraverso un parametro "embed" presente all'interno della query è possibile far riferimento ai campi di interesse. Per esempio, se vogliamo ottenere alcuni dettagli aggiuntivi riguardanti un determinato ticket, piuttosto che eseguire differenti richieste possiamo eseguirne una unica come la seguente:

    GET /tickets/12?embed=customer.name,assigned_user

    I campi che si vogliono ottenere sono separati da una virgola ed attraverso la "dot-notation" si possono ottenere i sotto-campi. Una possibile risposta alla richiesta sopra citata potrebbe essere:

```
{
    "id" : 12,
    "subject" : "I have a question!",
```

```
        "summary" : "Hi, ....",
        "customer" : {
            "name" : "Bob"
        },
        assigned_user: {
            "id" : 42,
            "name" : "Jim"
        }
    }
```

19. E' sempre consigliabile inserire controlli di "rate limit" all'interno dell'API. A questo proposito è possibile utilizzare il codice di stato HTTP 429 (Too Many Requests). Eventualmente, è possibile informare l'utente che sta per superare i limiti di utilizzo oppure quando potrà nuovamente effettuare delle richieste.

20. Le API RESTful devono essere "stateless", ovvero senza stato. Ciò significa che l'autenticazione non deve basarsi sui concetti di cookie o di sessione, ma ogni richiesta deve contenere delle credenziali di autenticazione. I token di autenticazione utilizzati da SSL sono una valida alternativa per semplificare l'autenticazione tramite credenziali.

21. L'utilizzo di un meccanismo di caching è semplificato dalla presenza

di un apposito framework all'interno di HTTP. In questo modo è solamente necessario includere nelle risposte alcuni header addizionali ed effettuare un controllo ulteriore per validare gli header delle richieste in ingresso. In Jersey è possibile effettuare i controlli sulla validità della cache attraverso un header predisposto "Cache-Control", ad esempio, il codice Cache-Control: max-age=10 setta la validità del valore max-age in 10 secondi.

```
@GET
@Produces("text/plain")
public Response get() {
    CacheControl cc = new CacheControl();
    cc.setMaxAge(10);
    return Response.ok("Some
        Data").cacheControl(cc).build();
}
```

Vi è anche la possibilità di rivalidare eventualmente la cache.

22. Per rappresentare ed informare l'utente di eventuali errori, l'API dovrebbe sempre ritornare un codice di stato HTTP, che può essere relativo ad errori client (serie di codici 400) o ad errori server (serie di codici 500). Gli errori (soprattutto quelli client-side) dovrebbero essere resi standard attraverso una rappresentazione JSON

dell'errore.

All'interno del progetto, ciò viene già gestito. Un possibile esempio è:

```
Response resp = null;
  try {
     final UUID modelID =
         this.hadrian.uploadModelPfa(fileDisposition
         .getFileName(), file, description);
     final URI modelURI =
         uriInfo.getAbsolutePathBuilder()
         .path(modelID.toString()).build();
     resp = Response.created(modelURI)
           .entity("Model " +
              fileDisposition.getFileName() + "
              uploaded successfully").build();
  } catch (IOException e) {
     resp =
         Response.status(Status.INTERNAL_SERVER_ERROR)
         .entity(e.getLocalizedMessage())
         .type("text/plain")
         .build();
  }
  return resp;
```

Capitolo 4

Continuous integration

Per Continuous integration si intente una pratica di sviluppo software dove gli sviluppatori integrano regolarmente (anche più volte al giorno) il proprio lavoro, con l'obiettivo di ottenere un software che possa essere rilasciato in tempi brevi e di individuare potenziali errori o problemi.

A questo proposito, vi sono diversi strumenti e metodologie che facilitano lo sviluppo del software.

Primo fra tutti, il DVCS git (Distributed Version Control System) permette di mantenere sincronizzati gli sviluppi delle singole feature di un progetto con la linea di progetto principale, consentendo inoltre di tener traccia (e quindi ripristinare) i singoli cambiamenti che vengono effettuati.

Le best practices per l'utilizzo di git consigliano di preferire l'utilizzo

del terminale rispetto alle interfacce grafiche del DVCS e di configurare il file .gitignore prima di iniziare lo sviluppo. Inoltre, è auspicabile non tenere traccia di file generabili (file binari, librerie, risorse e documentazioni generabili) per non causare un aumento eccessivo della dimensione del repository. Un ulteriore configurazione alla quale è necessario porre attenzione riguarda il comportamento in presenza di "newlines", in quanto inconsistente tra Windows e UNIX.

Per Git Flow si intende un flusso di sviluppo che descrive un modello di sviluppo costruito intorno al concetto di release del software. All'interno del progetto devono essere presenti diversi branch, tra i quali:

- il branch master, corrispondente alla mainline;
- il branch develop, che deriva da master e incorpora lo sviluppo;
- i feature branch, derivanti da develop con l'obiettivo di introdurre una nuova funzionalità;
- il branch release, derivante da develop con lo scopo di creare una nuova versione del progetto;
- i branch hotfix, derivanti da master con lo scopo di correggere un bug presente nella mainline del progetto e creare una nuova versione.

Ogni progetto va mantenuto in un repository. GitHub consente di ospitare il repository del progetto fornendo un supporto per le azioni

di fork, pull-request e per la revisione del codice. Ogni sviluppatore dovrebbe mantenere una copia (fork) del progetto presente nel repository centrale (truth repository) e richiedere a chi lo gestisce (attraverso delle pull-request) di unire il singolo sviluppo al progetto principale.

Solitamente, ogni software utilizza, oltre alle librerie di base, anche altre risorse esterne che possono dipendere da altri software. Essendo le dipendenze spesso molto numerose, può diventare difficile ricercarle, ottenerle, aggiornarle e verificarne manualmente la compatibilità.

Gradle è uno degli strumenti che ottimizzano la gestione delle dipendenze. Inoltre consente di compilare il software ed eseguirne i test, di generare documentazione e report. Gradle fornisce un supporto ottimale per diversi linguaggi, build incrementali ed esecuzione parallela di task. Attraverso uno script è possibile configurarne il comportamento.

I software che promuovono la continuous integration prevedono di eseguire una build ad ogni cambiamento del progetto, installando ogni volta un nuovo ambiente pulito dove eseguire le operazioni. Tra questi, Travis CI è uno strumento basato sul Web e ben integrato con GitHub: consente di mostrare i risultati delle build sul repository e di eseguire automaticamente una nuova build ogni volta che si presenta una pull-request. E' possibile inoltre configurare dei "cronjobs" che eseguano la build automaticamente dopo un certo periodo di tempo.

Lo script di continuous integration utilizzato all'interno del progetto in un ambiente Travis CI è:

```
sudo: required
dist: trusty

before_install:
 - chmod +x gradlew

language: java
jdk: oraclejdk8

script:
- './gradlew clean build'

services:
  - redis-server
```

Capitolo 5

Postman

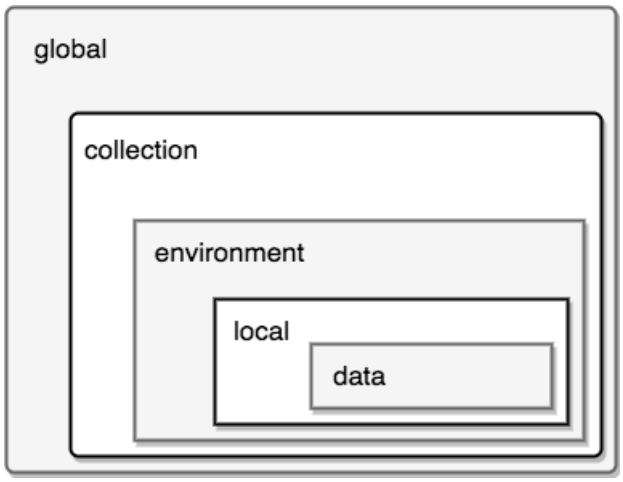
Postman è un'applicazione che consente di costruire, testare e documentare API più velocemente. Tramite Postman è infatti possibile effettuare delle chiamate API senza dover mettere mano al codice dell'applicazione, in quanto fornisce un'utile interfaccia grafica. Le chiamate possono essere effettuate sia verso un server locale che verso un server online impostando tutti i dati di una tipica chiamata API, dagli headers al body. Permette inoltre la creazione di ambienti in cui settare variabili e la possibilità di eseguire degli script, più o meno complessi, per testare le chiamate ed il comportamento delle API. E' anche possibile integrare diversi plugin in Postman fra cui Newman, che permette l'automazione dei test, e Interceptor per la personalizzazione degli header, la gestione dei cookies e la cattura delle richieste inviate tramite Chrome.

5.1 Environments

Un environments altro non è che un insieme di coppie chiave-valore in cui la chiave rappresenta il nome della variabile.

Postman permette l'utilizzo di variabili per poter salvare valori e riutilizzarli all'occorrenza in maniera molto semplice, ad esempio possiamo salvare in una variabile "dominio" l'indirizzo su cui andranno eseguite le richieste per poi utilizzare la sintassi dominio all'interno dell'URL della richiesta per inserirvi direttamente il nome dell'host.

Utilizzando gli script visti nel capitolo precedente si possono settare o prendere i valori delle variabili utilizzandoli in varie richieste. In Postman lo scope delle variabile viene gestito tramite la seguente gerarchia:



Se una variabile utilizzata nell'environment attivo condivide il nome con una variabile globale, la variabile dell'environment ha la priorità (contra-

riamente a ciò che avviene in altri sistemi). L'utilizzo degli ~~enviroments~~ diventa molto utile in quanto spesso è necessario, ad esempio, testare l'applicazione sia in locale che in remoto: è possibile creare due differenti environment con settaggi differenti così da passare da uno all'altro senza dover modificare ogni volta i parametri. Inoltre è possibile condividere, scaricare o importare un environment per poterlo condividere con un eventuale cliente o team di sviluppo.

Le variabili dichiarate all'interno dell'environment possono essere utilizzate, come detto in precedenza, attraverso il comando

{{nome_variabile}}

settate all'interno degli script con la funzione

pm.environment.set("nome_variabile","valore");

oppure, ancora, ne può essere preso il valore attraverso la funzione

pm.environment.get("nome_variabile");

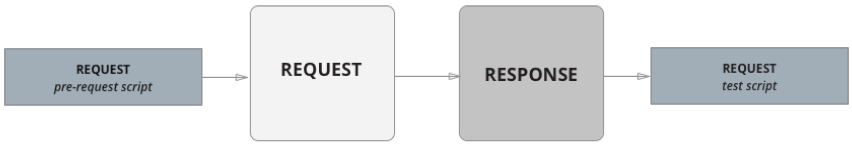
5.2 Script

In Postman è possibile scrivere e utilizzare script Javascript in due modi:

1. Con un **pre-request script**, prima che una richiesta venga inviata al server.

2. Con un **test script**, utilizzabile dopo la ricezione di una risposta.

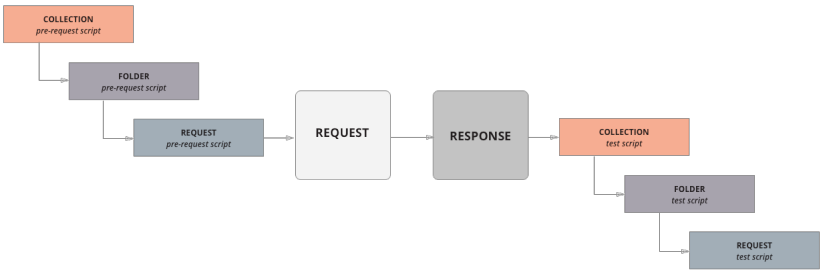
L'ordine di esecuzione degli script per una singola richiesta è riassumibile nello schema seguente:



In Postman, grazie alle collection è possibile raggruppare diverse richieste ed organizzarle in cartelle.

5.2.1 Pre-request script

Gli script di pre-request associati a una collection o a una cartella verranno eseguiti prima di ogni richiesta presente nella collection o nella cartella. Gli script di test associati a una collection o a una cartella verranno invece eseguiti dopo ogni richiesta presente nella collection o nella cartella.



Per ogni richiesta presente in una collection, gli script (sia di pre-request sia di test) verranno sempre eseguiti gerarchicamente: prima gli script a livello della collection, poi gli script a livello della cartella ed infine gli script a livello della richiesta.

Gli script di pre-request sono frammenti di codice associati a una richiesta che vengono eseguiti prima che la richiesta stessa sia inviata.

Ad esempio, all'interno del progetto è stato inserito il seguente pre-request script:

```
pm.sendRequest('http://localhost:9998/hadrian/models?
offset=0&limit=10', function (err, res) {
var result = res.json();
if(result[0] !== undefined){
    pm.environment.set("ID_download", result[0].id);
    console.log(pm.environment.get("ID_download"));
} else {
    console.log("Nessun modello caricato");
}
});
```

Il codice viene eseguito prima della richiesta di download di un determinato modello. Inizialmente viene inviata una richiesta che restituisce la lista dei modelli caricati, poi viene controllato che la lista non sia vuota ed in quel caso si richiede il download del primo modello presente nella lista.

Ciò è stato possibile utilizzando una variabile d'ambiente (ID_download) a cui viene assegnato il valore dell'ID del modello.

La chiamata finale è perciò:

```
GET http://localhost:9998/hadrian/models/pfaFiles/{{ID_download}}
```

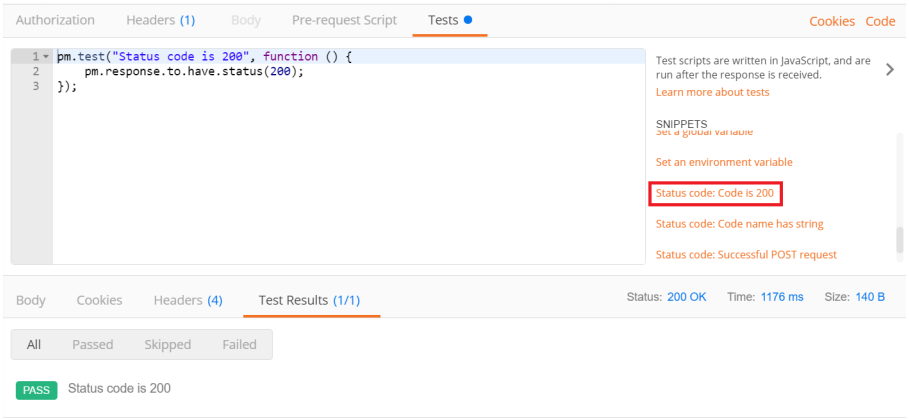
5.2.2 Test script

Gli script di test consentono di eseguire dei test per ogni richiesta che viene effettuata. Quindi, un test di Postman è essenzialmente composto da codice Javascript che viene eseguito dopo l'invio della richiesta e che utilizza l'oggetto *pm.response* per accedere alla risposta.

L'interfaccia di Postman fornisce alcuni test preimpostati direttamente di fianco all'editor. In questo modo è possibile aggiungere velocemente alcuni test che verranno eseguiti ad ogni richiesta.

I risultati sono mostrati nel tab Tests posizionato sotto il response viewer.

L'immagine seguente mostra un semplice test di controllo sullo stato della risposta:



E' possibile aggiungere script di test a una collection, una cartella, oppure a una singola richiesta presente in una collection. Uno script associato a una collection o a una cartella verrà eseguito dopo ogni richiesta presente nella collection o nella cartella, in modo da riutilizzare i test più comuni.

5.3 Testing

5.3.1 Newman

Newman è un tool a linea di comando per Postman basato su Node.js, che permette di eseguire una collection, quindi un'insieme di richieste e script, in maniera automatica visualizzando i risultati dei test e delle richieste. Una volta installato Node.js e Newman sul dispositivo è sufficiente lanciare il comando

```
$ newman run mycollection.json
```

L'esecuzione della collection è gestibile attraverso l'uso di flag all'interno della chiamata, ad esempio -h permette di visualizzare la guida con l'elenco delle opzioni disponibili, attraverso –delay-request [number] è possibile settare un delay per una data request o, similmente, con –timeout-request [number] settare un tempo massimo di esecuzione, -n specifica quante volte eseguire la collezione e così via. E' possibile anche gestire, per ogni esecuzione, un set di dati differenti semplicemente salvandoli su un file .json (questo permette di automatizzare, ad esempio, i test prima su un server locale e poi in remoto), oppure con dati differenti, senza dover ogni volta modificare le variabili interne alla collection.

Nel progetto preso in esame abbiamo creato una collection formata dalle seguenti richieste:

- uploadModelPfa
- listModels-xml
- downloadModelPfa
- deleteModel
- modelDescriptor-xml

Per *uploadModelPfa* è stato inserito il seguente test:

```
pm.test("Status code is 201", function () {
    pm.response.to.have.status(201);
    var str = postman.getResponseHeader("Location");
    str = str.split("/");
    pm.environment.set("ID_download", str[str.length - 1]);
    console.log(pm.environment.get("ID_download"));
});
```

Controlliamo che lo stato della risposta sia 201 (Created), poi settiamo la variabile d'ambiente "ID_download" con l'ID del modello appena caricato restituito dalla risposta.

Per *listModels-xml*:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
    var response = pm.response.json();
    console.log(response);
    pm.expect(response.length).to.be.above(0);
    var control = 0;
    for (var i = 0; i < response.length; i++) {
        if (response[i].id == pm.environment.get("ID_download"))
            {
            control++;
        }
```

```
    }
    pm.expect(control).to.equal(1);
    console.log(control);
});
```

In questo caso controlliamo che lo stato della richiesta sia 200 (OK), quindi controlliamo che l'elenco restituito non sia vuoto e che ci sia il modello caricato in precedenza.

Per *downloadModelPfa*:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
    var response = pm.response.json();
    console.log(response);
    pm.expect(response).not.to.be.null;
});
```

Controlliamo che lo stato della risposta sia 200 (OK) e che la risposta non sia nulla.

Per *deleteModel*:

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
    pm.sendRequest("http://localhost:9998/hadrian/models/
    {{ID_download}}", function(err, res) {
```

```
        console.log(res);
        pm.expect(res.code).to.equal(404);
    });
});
```

In questo test controlliamo che lo stato della richiesta sia 200 (OK) e proviamo a fare una richiesta sulla risorsa appena cancellata, aspettandoci risposta 404 (Not Found).

Creati tutti i test esportiamo la collection in formato Json; lanciando il comando sopra citato da console sul file appena esportato vengono effettuati tutte le richieste ed i relativi test in sequenza dando come risultato:

	executed	failed
iterations	1	0
requests	6	0
test-scripts	4	0
prerequest-scripts	2	0
assertions	4	0
total run duration: 874ms		
total data received: 2.94KB (approx)		
average response time: 106ms		

Fonti

1. Standard Services for Distributed Systems: Web Services, Andrea Omicini after Andrea Santi, DISI Università di Bologna, sede di Cesena, 2016
2. Engineering Distributed Systems - Service-Oriented Architectures, Alessandro Ricci, DISI Università di Bologna, sede di Cesena, 2017
3. Continuous integration and delivery, Danilo Pianini, Università di Bologna, sede di Cesena, 2017
4. Jersey Documentation (`https://jersey.github.io/`)
5. Postman Documentation (`https://www.getpostman.com/docs/v6/`)
6. `http://www.html.it/pag/19596/i-principi-dellarchitettura-restful/`
7. `https://skillsandmore.org/rest-e-restful-api-introduzione/`
8. `https://www.tutorialspoint.com/restful/index.htm`

9. https://www.ibm.com/developerworks/library/ws-restful/index.html

10. https://medium.com/@schneidenbach/restful-api-best-practices-and-common-pitfalls-7a83ba3763b5

11. https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

12. https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/

13. https://blog.philipphauer.de/restful-api-design-best-practices/